::ODMA\MHODMA\iManage;237379;1
JMS/GMB/dar
November 14, 2001

PATENT APPLICATION
Docket No.: 0050.2025-001

Date: 11/14/01 Express Mail Label No. EV010603181 US

Inventors: Jeffrey P. Grossman, Thomas F. Knight, Jr., Jeremy H. Brown and Andrew S. Huang

Attorney's Docket No.: 0050.2025-001

MECHANISM TO REDUCE THE COST OF FORWARDING POINTER ALIASING

RELATED APPLICATION

This application claims the benefit of U.S. Provisional Application No. 60/299,244, filed on June 19, 2001.

The entire teachings of the above application are incorporated herein by reference.

GOVERNMENT SUPPORT

The invention was supported, in whole or in part, by a grant F30602-98-1-0172 from Air Force Research Lab. The Government has certain rights in the invention.

BACKGROUND OF THE INVENTION

Forwarding pointers are an architectural mechanism that allow references to a memory location to be transparently forwarded to another location. Known variously as "invisible pointers," "forwarding pointers" and "memory forwarding," they are familiar to the hardware community but to date have been incorporated into very few architectures.

One reason that forwarding pointers have received little support is that they have been perceived as possessing limited utility. Recently, however, it has become apparent that forwarding pointers are indeed useful constructs that can expedite program execution. Chi-Keung Luk and Todd C. Mowry, "Memory Forwarding: Enabling

Aggressive Layout Optimizations by Guaranteeing the Safety of Data Relocation," Proc. ISCA 1999, pp. 88-99 (hereafter "Luk"), incorporated by reference herein in its entirety, show that using forwarding pointers to perform safe data relocation can result in significant performance gains on arbitrary programs written in C, speeding up some applications by more than a factor of two. Jeremy Brown, "Memory Management on a Massively Parallel Capability Architecture", Ph.D. thesis proposal, M.I.T., December 1999, gives an algorithm for performing asynchronous local compacting garbage collection in a massively parallel distributed system. This algorithm uses forwarding pointers to avoid the high run-time costs usually associated with such a system. Thus, there is growing motivation to include hardware support for forwarding pointers in novel architectures.

A second and perhaps more significant reason that forwarding pointers have received little attention from hardware designers is that they introduce aliasing - that is, it is possible for two different pointers to resolve to the same word in memory.

Fig. 1A illustrates how aliasing occurs. A first pointer P_2 14 points directly to some target data 16. A second, indirect pointer, P_1 10 points to a forwarding pointer 12 which in turn points to the target data 16. Thus, pointers P_1 and P_2, which hold different values, resolve to the same target data 16.

Figs. 1B and 1C illustrate one manner in which such a scenario can occur. Here, two data objects A 4 and D 16 are stored in a memory 2A (Fig. 1B), with a pointer P_1 10 directly referencing data object D 16. As a result of data compaction or other operations, data object D 16 is moved, as shown in memory 2B (Fig. 1C), and a new forwarding pointer 12 is inserted into data object D's old location. Thus pointer P_1 10 now points to the forwarding pointer 12 which points to the new location of data object D 16. Pointer P_1 10 is therefore now an indirect pointer. Meanwhile, a new direct pointer P_2 14 has been created which points to the new location of data object 16, resulting in the combination of pointers pictured in Fig. 1A.

The presence of this aliasing necessarily introduces run time costs in order to ensure correctness of execution. In Luk, two specific problems are identified. First, direct pointer comparisons are no longer a safe operation; some mechanism must be provided for determining the final addresses of the pointers. Second, seemingly independent memory operations may no longer be reordered in out-of-order machines.

In Luk, the problem of pointer comparisons is addressed by inserting code to determine the final address for each pointer, unless the compiler is able to determine that the pointers do not point to relocated objects. The overhead of this approach is potentially large.

In the best case, both target memory words will be resident in the cache, neither of them will contain a forwarding pointer, and the pointer comparison will be slowed down by roughly an order of magnitude. However, since pointer comparisons often precede a decision to perform operations on an object, a common case will be when one or both dereferences cause a cache miss, slowing down the comparison by another order of magnitude.

The solution proposed by Luk for reordering memory operations is to use "data dependence speculation," which allows loads to execute speculatively before it is known that they are independent of any preceding stores. In an architecture that supports data dependence speculation, it is fairly easy to extend the hardware to operate correctly in the presence of forwarding pointers. Luk found that this solution is effective as incorrect speculation occurs only rarely. However, Luk assumes the presence of some fairly complex hardware. For architectures in which silicon area efficiency is a concern, a lower cost alternative is preferable.

The forwarding pointer aliasing problem is an instance of the more general challenge of determining object identity in the presence of multiple and/or changing names. This problem has been studied explicitly. See, for example, Setrag N. Khoshafian, George P. Copeland, "Object Identity", Proc. 1986 ACM Conference on Object Oriented Programming Systems, Languages and Applications, pp. 406-416, incorporated by reference herein in its entirety.

A natural solution which has appeared time and again is the use of system-wide unique object IDs or UIDs. UIDs completely solve the aliasing problem, but have two disadvantages.

First, the use of UIDs to reference objects requires an expensive translation each time an object is referenced to obtain the virtual address of the object.

Second, quite a few bits are required to ensure that there are enough UIDs for all objects and that globally unique IDs can be easily generated in a distributed computing environment. In a large system, at least sixty-four bits would likely be required in order to avoid any expensive garbage collection of UIDs and to allow each processor to allocate UIDs independently.

SUMMARY OF THE INVENTION

In the worst case, one of the pointers points to data which is not even resident in main memory. This can occur frequently in programs that deal with massive datasets. It is desirable to be able to compare pointers without having to dereference them. By using short quasi-unique identifiers (SQUIDs), it is possible to do so in the majority of cases.

SQUIDs maintain one of the advantages of UIDs, namely they allow pointers to different objects to be distinguished quickly, and with high probability. Because SQUIDs are not unique, they can be much shorter than UIDs, comprising only a small number of bits, while still providing similar functionality. Furthermore, SQUIDs do not require any translation tables, since they are a part of the pointer format.

Therefore, in accordance with one aspect of the present invention, a data processing system comprises memory for storing data objects, where the data objects are referenced by pointers. A short-quasi-unique-identifier (SQUID) generator generates and assigns SQUIDs to data objects stored in the memory segment. Pointers to a particular data object contain the data object's assigned SQUID.

The system further comprises a memory allocator which allocates a segment of the memory to a data object. If the data object is moved to a second allocated memory segment, for example, due to resizing, data compaction or garbage collection, a new pointer to the second allocated memory segment is placed at the original memory segment, so that any pointers to the original memory segment now point to the new pointer. The data object might also be moved from a first memory to a second memory within a distributed system.

In at least one embodiment, the distribution of SQUIDs over a range is uniform. SQUIDs can be generated by counting, generated randomly, generating through some hashing mechanism, or other means.

Where two different pointers must be compared, a comparator compares their respective SQUIDs. The comparator determines that the two pointers do not reference the same data object if the SQUIDs are different. On the other hand, the comparator determines that the two pointers reference the same data object if the SQUIDs are identical and the address fields of the two pointers are identical.

In at least one embodiment, each pointer address field comprises a base address and an offset, and the comparator determines that the two pointers do not reference identical locations within a referenced data object if the pointers' offsets are not identical.

SQUIDs can be implemented either in hardware, or software, or a combination.

According to another aspect of the invention, a pointer is associated with a migration indicator field which indicates the number of migrations of the referenced data object prior to the pointer being created. The comparator determines that two pointers do not reference the same data object if their associated migration indicators indicate identical numbers of migrations and their corresponding addresses are different. The migration indicator can comprise just one bit, or may comprise multiple bits.

In at least one embodiment of the present invention, pointers are guarded pointers.

BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing and other objects, features and advantages of the invention will be apparent from the following more particular description of preferred embodiments of the invention, as illustrated in the accompanying drawings in which like reference characters refer to the same parts throughout the different views. The drawings are not necessarily to scale, emphasis instead being placed upon illustrating the principles of the invention.

Fig. 1A is a schematic diagram illustrating the concept of forwarding pointer aliasing.

Figs. 1B and 1C are schematic diagrams illustrating one manner in which the scenario of Fig. 1A can come about.

Fig. 2 is a block diagram of an embodiment of the present invention.

Fig. 3 is a schematic diagram illustrating a simple SQUID embodiment of the present invention.

Figs. 4A-4E are schematic diagrams illustrating the different scenarios described in Table 1.

Fig. 5 is a schematic diagram illustrating how the invention can be used in deciding whether to reorder instructions.

Fig. 6 is a schematic diagram illustrating an embodiment of the present invention that utilizes SQUIDs and a one-bit migration indicator.

Fig. 7 is a schematic diagram illustrating an embodiment of the present invention that utilizes SQUIDs and a multiple-bit migration indicator.

Fig. 8 is a schematic diagram illustrating an application of the present invention to guarded pointers.

DETAILED DESCRIPTION OF THE INVENTION

A description of preferred embodiments of the invention follows.

The present invention is a simple mechanism which can be used to reduce the cost of forwarding pointer aliasing. Each object is assigned a short random tag, to be stored in each pointer to the object. This tag is similar in role to a unique identifier

(UID), but is not necessarily unique. These tags are referred to as Short Quasi-Unique IDentifiers, or SQUIDs.

Fig. 2 illustrates an embodiment in which a memory 100 can hold many different data objects 102, two of which are shown. Unused memory space from the memory 100 is allocated to a new data object by a memory allocator 104 when the data object is first created. A SQUID is generated by a SQUID generator 106, and supplied to the memory allocator 104. The memory allocator 104 generates a pointer to the data object.

Thereafter, new pointers need by various applications 112 are generated by a pointer copier 114 that copies existing pointers. Since the SQUID is part of the pointer format, it is copied when an existing pointer is copied.

For example, when a device such as garbage collector 108 moves an object, it references the object through a pointer. When the object is moved, a forwarding pointer, pointing to the object's new location and generated for example by a pointer generator 110, is left at the old location. The SQUID from the referencing pointer is copied by the pointer generator 110 into the forwarding pointer.

In the common case, SQUIDs allow pointer comparison and memory operation reordering to proceed with no overhead. Only in rare cases is it necessary to degrade performance to ensure correctness. Thus, SQUIDs allow an architecture to support forwarding pointers with reduced average run-time overhead. Furthermore, this overhead can be eliminated altogether if the software chooses not to make use of forwarding pointers.

As shown in Fig. 3, to implement SQUIDs, a pointer 20 according to the present invention comprises an address field 21 augmented with a short n-bit tag field 23, where n is, for example, from eight to sixteen bits in length. This tag field 23 is filled with a SQUID which is assigned a value when an object is allocated. Preferably, SQUIDs are assigned according to a uniform distribution over some range. Such assignment could be implemented, for example, by a simple counting function, randomly, or by a hashing function.

Unlike a UID, two pointers with the same SQUID might not point to the same object. However, two pointers with different SQUIDs necessarily point to different objects. In many cases this fact alone can be used to avoid the run-time costs of forwarding pointer aliasing.

Pointer Comparisons

An address is typically logically divided into a base 21A and an offset 21B. Two pointers can be efficiently compared by examining their base addresses, offsets and SQUIDs. Table 1 below illustrates the different possible scenarios.

In CASES 1 and 2, if the base addresses are the same, then the pointers point to the same object, and the pointers are the same if and only if they have the same offset into the object.

Fig. 4A illustrates CASE 1. Two pointers 60, 62 have identical SQUIDs, base addresses and offsets. Therefore, they both pont to the same data at address 104 in data object, Object A 66 within the memory 64.

Fig. 4B illustrates CASE 2, in which only the offsets of the two pointers 60, 62 are different. In this case, the pointers 60, 62 both point to the same object, Object A 66, but they point to different data, pointer 60 pointing to data located at address 104 and pointer 62 pointing to data located at address 132

As Fig. 4C illustrates, if, as in CASE 3, the SQUIDs are different, then the pointers 60, 62 point to different objects, here shown as Object A 66 and Object B 68..

It is only in CASE 4, that is, in the case that the base addresses are different but the SQUIDs and offsets are the same, that it is necessary to perform expensive dereferencing operations to determine whether or not the final addresses are equal.

	SQUIDS	BASES	OFFSETS	POINTERS
CASE 1	SAME	SAME	SAME	SAME
CASE 2	SAME	SAME	DIFF	Point to same object, but different offsets, so pointers are DIFFERENT
CASE 3	DIFF	X	X	DIFFERENT
CASE 4	SAME	DIFF	SAME	Must be resolved

Table 1

It can be argued that this latter case, i.e., CASE 4, will be rare. It occurs in two circumstances: either the pointers reference different objects which have the same SQUID, or the pointers reference the same object through different levels of indirection.

Fig. 4D illustrates a first scenario for CASE 4, in which the two pointer 60, 62 point to different objects, Object A 66 and Object B 68, whose SQUIDs have the same value, i.e., 23 in this example.

Fig. 4E illustrates the second scenario for CASE 4, in which the second pointer 62 indirectly points to the same object, i.e., Object A 66, as the first pointer 60, through a forwarding pointer 70.

The former scenario (Fig. 4D) occurs with probability 2^{-n}. The latter scenario (Fig. 4E) is application dependent, but we note that (1) applications tend to compare pointers to different objects more frequently than they compare pointers to the same object, and (2) the results of the simulations in Luk indicate that it is reasonable to expect the majority of pointers to migrated data to be updated, so that two pointers to the same object will usually have the same level of indirection.

Reordering Memory Operations

As Fig. 5 illustrates, in a similar manner, a decision can be made, most likely by hardware, as to whether or not it is possible to reorder memory operations based on

SQUIDs. Two pointers 120, 122 are compared by a comparator 124. An instruction reorderor 126 decides whether to reorder instructions based on the output of the comparator 124.

If the two pointers 120, 122 have different offsets or different SQUIDs then the corresponding operations can be safely reordered. If the offsets and the SQUIDs are the same, then the operations are not reordered. No other mechanism is required to guarantee correctness of execution, and the probability of failing to reorder references to different objects is 2^{-n}.

Improving Performance

The use of SQUIDs reduces the average overhead necessary to check for aliasing to a small but still non-zero amount. Ideally, pointers to objects which are never migrated should incur no overhead whatsoever.

As illustrated in Fig. 6, this can be achieved by adding a single 'migrated' bit (M) 25 to the pointer 30 to indicate whether or not the pointer points to the original address at which the object was allocated. When a new object is created, pointers to that object have M = 0. When the object is migrated, pointers to the new location (and all subsequent locations) have M = 1. If two pointers each with M = 0 are being compared (either as the result of a user comparison instruction, or to determine whether or not memory operations can be reordered), the SQUIDs can be ignored and the comparison performed based on the addresses alone, and in fact, this migration bit 25 can be used without SQUIDs. Hence, there is no runtime cost associated with support for forwarding pointers until the software makes use of them.

As illustrated in Fig. 7, the migration indicator 27 can be expanded to multiple bits to indicate the migration generation, that is, the number of times the data has moved Pointers having migration indicators 27 that are equal (but not saturated - see below) can be compared without reference to the SQUIDs. When comparing two pointers having different values in their respective migration indicators, it is only necessary to dereference one of the pointers until the migration indicators match, at which point a valid address comparison can be performed. Of course, a pointer's migration indicator must saturate at some maximum value determined by its length, at which point it is not valid to use the migration indicator to compare pointers.

Hardware and Software Overhead

The only software overhead required to support SQUIDs is the code that generates them when objects are allocated. This adds just a few instructions to memory allocation. A trap handler is needed to check for aliasing when comparing different addresses with the same SQUID, but this code (or an equivalent hardware mechanism) is a general requirement for supporting forwarding pointers without UIDs and is not specific to the implementation of SQUIDs. Moreover, placing a single copy of this code in a trap handler creates much less software overhead than inlining the code at every pointer comparison as suggested by Luk.

In order to be effective, SQUIDs require only a small number of bits to be added to the pointer. For example, if eight bits are added (seven SQUID bits and one migrated bit), then the probability of failing to distinguish pointers to different objects is less than 0.008. The hardware required to implement SQUIDs consists of some simple logic to inspect SQUID/M bits for pointer comparisons and memory operation reordering, and support for a trap which occurs when different pointers with the same SQUIDs are compared.

SQUIDs can be applied to guarded pointers, which are described in U.S. Patent No. 5,845,331 to Carter et al. (hereafter "Carter"), which is incorporated by reference herein in its entirety. Guarded pointers are a form of unforgeable capabilities, which include both a pointer and segment information within the guarded pointer itself. Guarded pointers are discussed in R.S. Fabry, "Capability-Based Addressing", Communications of the ACM, Volume 17, Number 7, pp. 403-412, July 1974, which is incorporated by reference herein in its entirety. The specific format of the guarded pointer is not important, but we assume that it is possible to determine the base address of an object given a pointer to the object's interior, as in Carter.

Fig. 8 illustrates a guarded pointer embodiment of the present invention in which the guarded pointer 50 consists of an address 51, segment information 53, and a single pointer bit P 59 to distinguish guarded pointers from data. To support SQUIDs, a single migrated bit M 57 and a small number of SQUID bits (seven as shown) 55 are added.

Applications

Forwarding pointers are a key enabling mechanism for safe data compaction and efficient garbage collection. The present invention allows forwarding pointer support to be incorporated into novel architectures with little or no average run-time cost due to aliasing.

Luk made clear the advantages of data relocation in the context of a uniprocessor. By compacting live data, better use can be made of the cache and as a result program execution is sped up by as much as a factor of two. In a distributed shared memory multiprocessor it is also important to be able to relocate data for a different reason: a processor can access local memory an order of magnitude faster than it can access remote memory. Effective computation on such a machine therefore depends on being able to move data to the processing node at which it is needed. SQUIDs provide efficient support for data migration and are therefore applicable to both single processor and multiprocessor high performance systems.

Historically, one of the primary uses of forwarding pointers has been to implement incremental garbage collection. More recently, it is shown in Jeremy Brown, "Memory Management on a Massively Parallel Capability Architecture," Ph.D. thesis proposal, M.I.T., December 1999, incorporated by reference herein in its entirety, that forwarding pointers can be used to implement efficient local compacting garbage collection in a massively parallel distributed system. Hardware support for fast garbage collection is especially important given the growing prevalence of the Java programming environment, which is the language of choice for web programming and which specifies a garbage collected memory model. Another application of SQUIDs is therefore the implementation and/or improvement of systems which are specifically designed to run Java efficiently. Such systems are currently under development. For example, see Marc Tremblay, "An Architecture for the New Millenium", Proc. Hot Chips XI, Aug. 15-17, 1999, which is incorporated by reference herein in its entirety.

A SQUID need not necessarily be a part of the pointer itself. For example, Instead, a "SQUID cache" could be employed that stores SQUIDS of recently-used pointers. SQUIDS would then be retrieved by presenting the pointer address to the cache. While this might slow execution and require more complicated hardware, it might actually be a useful technique if the size of pointers is so severely constrained that there is absolutely no way to include the SQUID, for example, in a 32 bit machine in which all 32 bits are required for the pointer address.

While this invention has been particularly shown and described with references to preferred embodiments thereof, it will be understood by those skilled in the art that various changes in form and details may be made therein without departing from the scope of the invention encompassed by the appended claims.